Advanced Data Structure

Programming Project - Rising City

Name: Yu-Peng Chen

UFID: 7904-3193

UF E-mail Account: yupengchen@ufl.edu

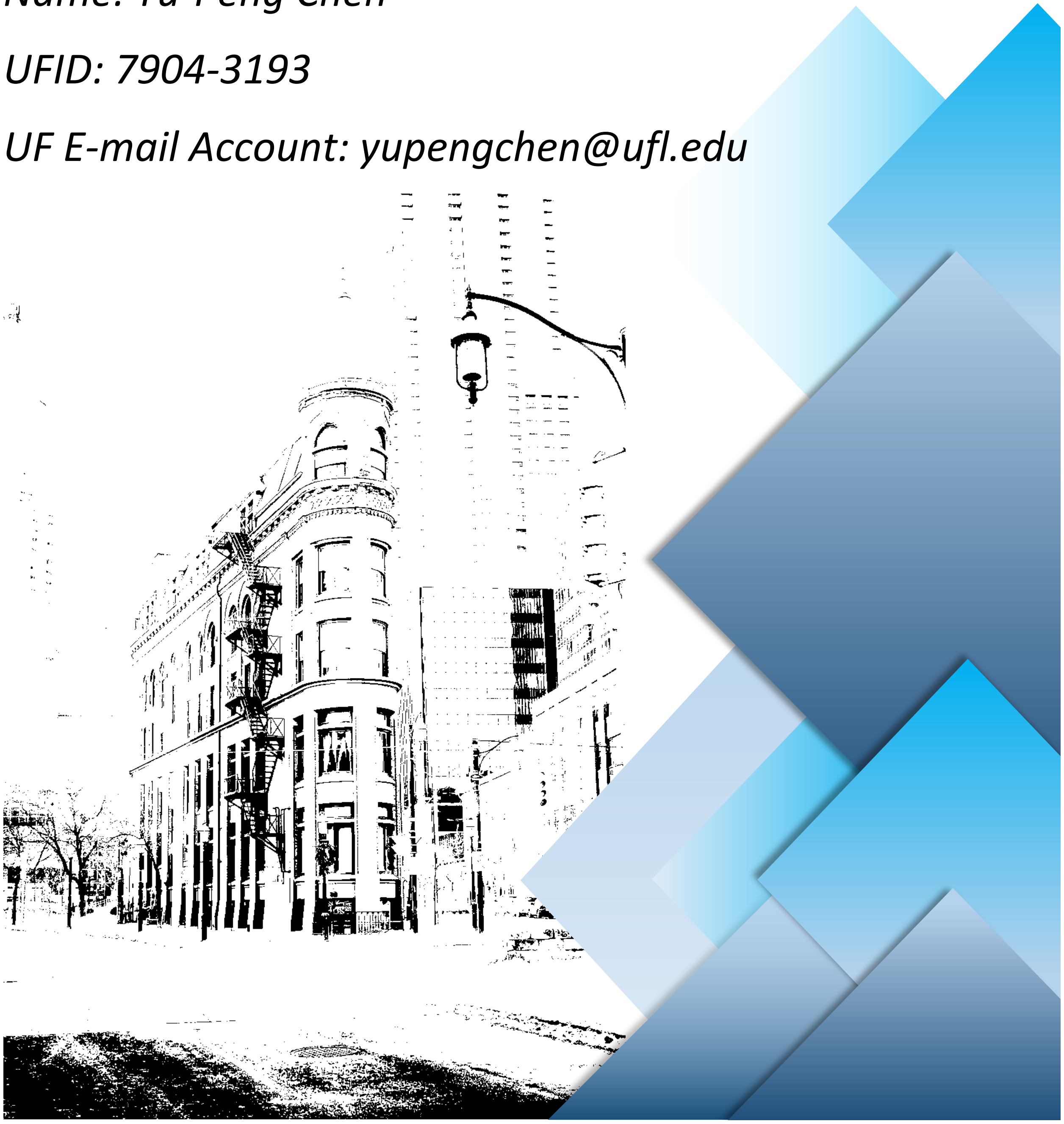
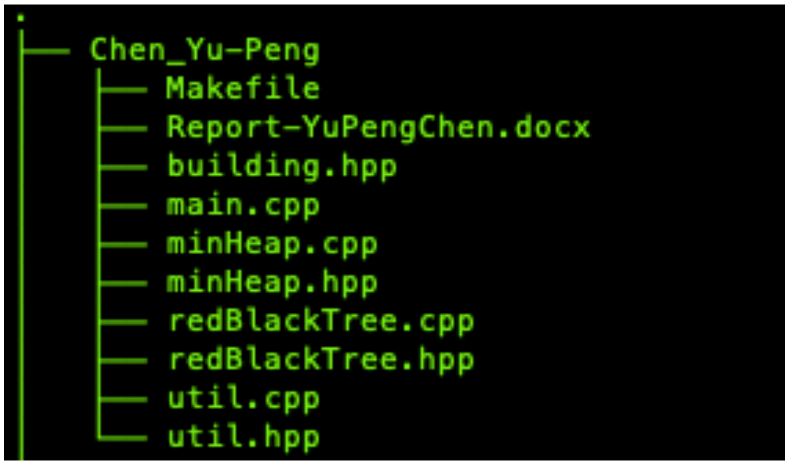


Table of Contents

1. Files



```
.
├── Chen_Yu-Peng
│   ├── Makefile
│   ├── Report-YuPengChen.docx
│   ├── building.hpp
│   ├── main.cpp
│   ├── minHeap.cpp
│   ├── minHeap.hpp
│   ├── redBlackTree.cpp
│   ├── redBlackTree.hpp
│   ├── util.cpp
│   └── util.hpp
```

Figure 1. File list within the directory

As shown in *Figure 1*, the following are the 10 files within the submitted directory:

- Makefile
- Report-YuPengChen.docx
- building.hpp
- main.cpp
- minheap.cpp
- minheap.hpp
- redBlackTree.cpp
- redBlackTree.hpp
- util.cpp
- util.hpp

2. Program Structure

1) Components

The components of this program can be divided into 3 parts as follows:

a. Main function

The entry point of the program. It also handles the essential elements as follows:

- Global Time
- Flags that decide the timing of insertion, printing, and constructing
  - buildingCity
    - F: stop building the city if no building is selected to work on.
    - T: keep on building the city.
  - workingOnBuilding
    - F: select building to work on.
    - T: working on a certain building, constructingTime > 0.
  - fileFlag
    - F: no commands are left to read in the file.
    - T: a file command is read successfully.
  - readFileLine
    - F: do not get lines from the input file.
    - T: get lines from the input file and parse it into arguments.
  - specialFlag
    - Use this flag to deal with the special case when
      printing and completing a building happen at the same time.
      (will be shown later in the report)
- Others
  - Min heap city
    - An array of buildings with maximum capacity = 2000.
  - RBT city
    - Initialize a memory space for a new building, and maintain the city using pointers in the RBT.
  - Selected building
    - The building selected to work on using extractMin() from min heap.
  - Constructing time
    - Use it to keep track of the executed time in one round. Max = 5.

b. Data structures

- Min heap
  - Use it to **select** the building with the least executed time.

- Red black tree
    - Use it to **search** for a certain building
- Building
    - The triplet: (buildingNum, executedTime, totalTime)

c. Utility functions

- Read command
    - A parser for the command in the input file.
- Select building
    - Use min heap to select the building with the least executed time to work on.
- Construct building
    - Work on the selected building for one day at a time.
- Insert building
    - Insert a new building into min heap, initialize another space for the same building, and insert the pointer pointing to that space into RBT.
- Print building
    - Print(buildingNum1) or Print(buildingNum1, buildingNum2)

2) Program Flow

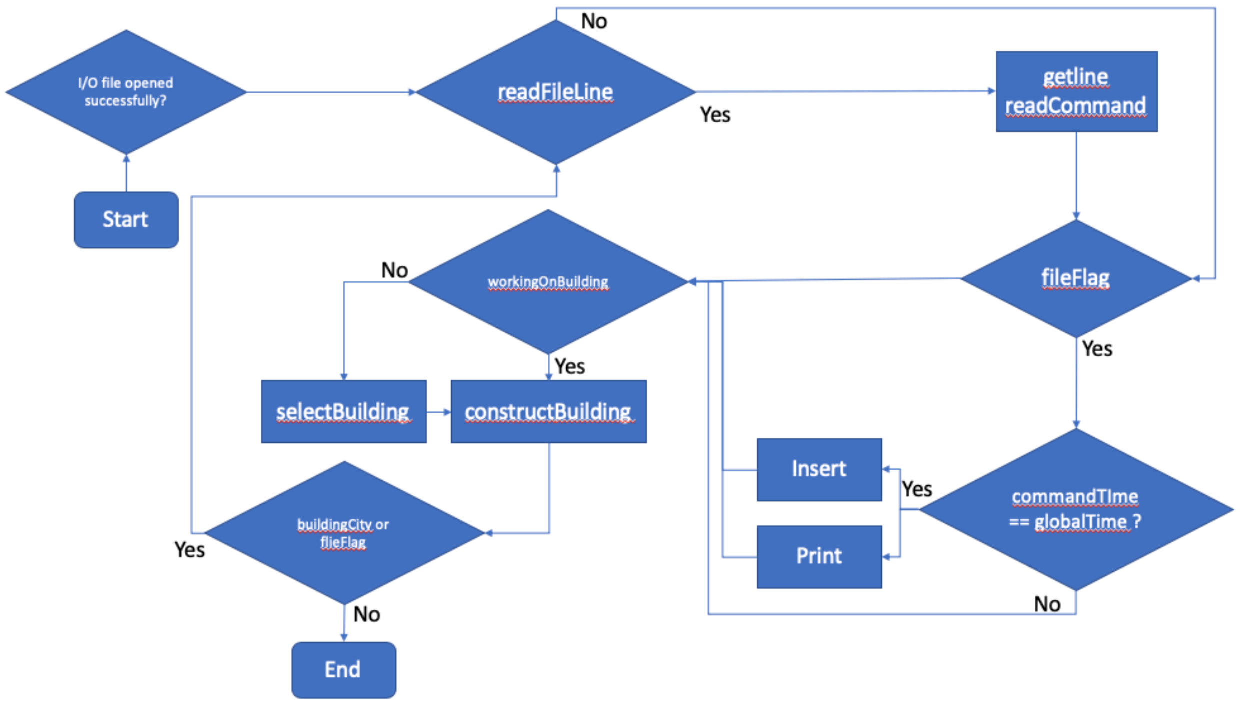


Figure 2. Program flow.

As shown in *Figure 2.*, there are several decisions to make:

a. readFileLine
   If true, the next command in the file will be read using **getline** and **readCommand**. Otherwise, we will move on to the next decision point.
b. fileFlag
   True for this flag means that there are still unread commands in the input file, and it also means that currently there is a command that is about to be executed or waiting for execution.
c. commandTime and globalTime
   If commandTime equals to globalTime, the command will be executed. Otherwise, **readFileLine** will be set to false, which means that we wait for the command to be executed before getting the next command in the input file.
d. workingOnBuilding
   If true, we skip the selecting process and keep on constructing the building that is under construction using the **constructBuilding()** function. Otherwise, **selectBuilding()** is executed first and then the **constructBuilding()** function
e. buildingCity
   This flag is always true until the nullBuilding is return from the min heap

```
building nullBuilding;
nullBuilding.buildingNum = -1;
nullBuilding.executedTime = INT_MIN;
nullBuilding.totalTime = INT_MAX;
```

3. Classes and Functions

1) util.cpp

- insertBuilding(…)
  - Insert the selected building into the min heap using the **insertKey()** function.
  - Insert the selected building into the RBT using the **rbtInsert()** function.

- printBuilding(buildingNum, …)
  - Uses the **treeSearch()** function to search the RBT for the building that is going to be printed.
  - The program only updates a building’s executedTime when the building has been built for 5 days or completed. Therefore, when the program is asked to print the building that is still under construction, we have to add the current constructingTime to the building returned from **treeSearch()**.
  - Time: O(log(n))

- printBuilding(buildingNum1, buildingNum2, …)
  - Uses the **printingRootSearch()** function to search for the root of the subtree that covers the range of [buildingNum1,buildingNum2].
    - Time O(log(n))
  - Uses the **printInorder()** function to print the buildings using inorder tree traversal.
    - Time O(S)
  - Handles the punctuations using the **delimiterFlag** flag.
  - Time: O(log(n) + S)

- selectBuilding(…)
  - Uses the **extractMin()** to select the root of the min heap as the building to work on.

- constructBuilding(…)
  - Increments the executedTime of the selected building.
  - If executedTime is equal to the total time, we update the executedTime in the RBT using the **updateTime()** function and set the **specialFlag** to true. The removal is performed in the main function.
  - If the building has been worked on for 5 days but is not completed, we use the **updateTime()** function to update the executedTime in the RBT and **insertKey()** function to insert the constructed building into the min heap.
  - Otherwise, set the **workingOnBuilding** flag true and return it.

2) minHeap.cpp

- minHeapify(…)
- insertKey(…)
- decreaseKey(…)
- extractMin()
  - return and remove the root
  - replace the root with the latest element in the array and do a **minHeapify()** on the root.
- deleteKey(…)
  - First uses the **decreaseKey(…)** function to decrease the deleting key the INT_MIN, and then uses the **extractMin()** function to extract it

3) redBlackTree.cpp

- leftRotate(…)
  - Do RR rotation
- rightRotate(…)
  - Do LL rotation
- treeSearch(…)
  - If the buildingNum is not found in the tree, **the last visited node** will be returned.
- rbtInsert(…)
  - Do a **treeSearch(…)** and make the insert node left child or right child of the last visisted node.
  - Do **insertFix(…).**
- rbtDelete(…)
  - Uses **deleteNode(…)** to remove the building and **deleteFix(…)** to rebalance the tree.
  - If the deleting node is a 2-degree node, we replace it with the rightmost node of the deleting node's left child, which means the largest in the left subtree, by returning **rightMost(x->left).**
- printingRootSearch(…)
  - Use a binary tree search for the buildingNum that falls between buildingNum1 and buildingNum2 starting from the root. This way the root of the subtree that covers the range [buildingNum1,buildingNum2] can be found uniquely.
- updateTime(…)
  - Do a **treeSearch(…)** to find the building that is going to be updated, and then update the building's executedTime.

4. Special Cases

1) Tie Break

When it comes to select a building to work on, the building with the least executedTime is selected. Ties are broken (when two building have the same executedTime) by **comparing the buildingNum and choosing the smaller one** (buildingNum is a unique number). The following are functions that need to be changed to deal with this special case in the min heap data structure:

- minHeapify()
- insertKey()

- decreasKey()

The common part for these three functions is that there are comparisons between parents and children after executing the operation. The original comparison is implemented simply with a while loop as follows.

```
while(heapArr[idx].executedTime<heapArr[parent(idx)].executedTime && idx != 0){
    exchange(&heapArr[idx], &heapArr[parent(idx)]);
    idx = parent(idx);
}
```

Taking the tie breaks into consideration, the implementation is change slightly as follows:

```
while(heapArr[idx].executedTime<=heapArr[parent(idx)].executedTime && idx != 0)
{
        if(heapArr[idx].executedTime < heapArr[parent(idx)].executedTime){
            exchange(&heapArr[idx], &heapArr[parent(idx)]);
            idx = parent(idx);
        }
        else if(heapArr[idx].executedTime == heapArr[parent(idx)].executedTime)
        {
            if(heapArr[idx].buildingNum < heapArr[parent(idx)].buildingNum){

                exchange(&heapArr[idx], &heapArr[parent(idx)]);
                idx = parent(idx);
            }
            else{
                break;
            }
        }
}
```

In the above implementation, we consider both the cases when children's executedTime is **less than** and **equal to** the parent's executedTime. Thus, the while loop argument is slightly changed, and the following implementation is also changed correspondingly.

2) Printing and Completing a building at the same time

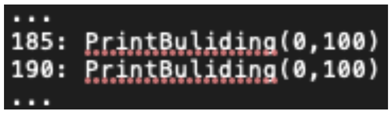
Initially, the program is designed to print to the output file the (buildingNum, completionDate) pair and remove the building from the tree before the input command is executed. The flow of my implementation is:

Construct the building → output (buildingNum, completionDate) pair and remove the building if it is completed → increment the global time → execute command read from the input file if global time equals the command time (global time read from the input file).

As a result, there will be different outputs when printing and completing a building happen at the same global time.

This case happens in the Sample_input2.txt file:

The input commands are:



```
...
185: PrintBuliding(0,100)
190: PrintBuliding(0,100)
...
```

And the original output for my implementation is:

```
(15,50,200),(30,45,50),(40,45,60),(50,45,100)

(30,190)

(15,50,200),(40,45,60),(50,45,100)
```

When the sample output provided is:

```
(15,50,200),(30,45,50),(40,45,60),(50,45,100)

(15,50,200),(30,50,50),(40,45,60),(50,45,100)

(30,190)
```

From the example we can see that building 30 is completed on the 190th day, which is also the day that PrintBuilding(0,100) should be executed.

To deal with this circumstance without changing the overall constructing flow, I added the **specialFlag** that is set to true when the building is completed, and use it to signal the removal of the completed building after printing. The resulting flow is slightly changed to:

*Construct the building → **specialFlag** = true if the building is completed → increment the global time → execute command read from the input file if global time equals the command time (global time read from the input file)*

Note that there are 3 cases to be considered after before executing the command:

a. if Insert:
→ output (buildingNum, completionDate) pair and remove the building if ***specialFlag*** *= true → Insert*

b. if PrintBuilding:
→ PrintBuilding → output (buildingNum, completionDate) pair and remove the building if ***specialFlag*** *= true*

c. Otherwise
→ output (buildingNum, completionDate) pair and remove the building if ***specialFlag*** *= true*

5. Makefile

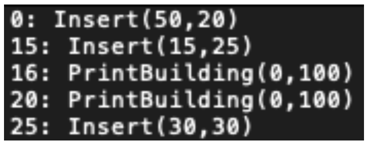
4) CC = g++
   - The g++ compiler is used.
5) EXEC = risingCity
   - Specify the name of the executable file.
6) OBJS
   - Object files: minHeap.o redBlackTree.o main.o util.o
   - New implementations should be added to this line.
7) all: ${EXEC}
   - Compile and produce risingCity.exe using the **make** command.
8) clean:

   ```
   rm -rvf *.o ${EXEC}
   ```
   - Clean object files and the executive file using the **make clean** command.

6. Sample Results

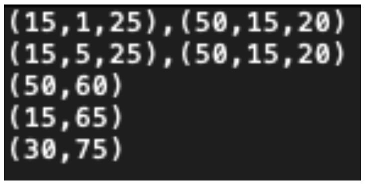
1) input.txt

- input:



```
0: Insert(50,20)
15: Insert(15,25)
16: PrintBuilding(0,100)
20: PrintBuilding(0,100)
25: Insert(30,30)
```

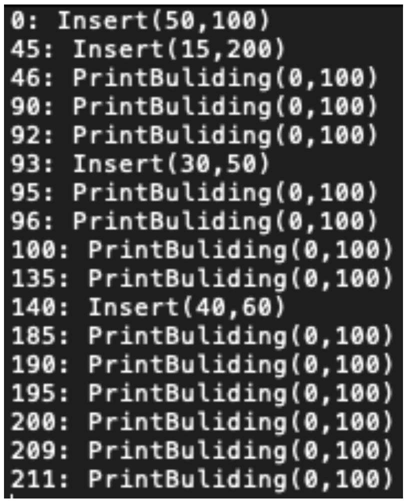
- output:



```
(15,1,25),(50,15,20)
(15,5,25),(50,15,20)
(50,60)
(15,65)
(30,75)
```

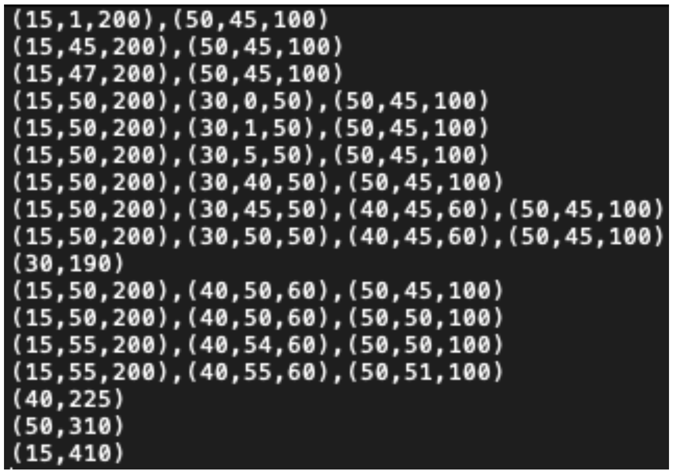
2) Sample_input2.txt

- input:



```
0: Insert(50,100)
45: Insert(15,200)
46: PrintBuliding(0,100)
90: PrintBuliding(0,100)
92: PrintBuliding(0,100)
93: Insert(30,50)
95: PrintBuliding(0,100)
96: PrintBuliding(0,100)
100: PrintBuliding(0,100)
135: PrintBuliding(0,100)
140: Insert(40,60)
185: PrintBuliding(0,100)
190: PrintBuliding(0,100)
195: PrintBuliding(0,100)
200: PrintBuliding(0,100)
209: PrintBuliding(0,100)
211: PrintBuliding(0,100)
```

- output:



```
(15,1,200),(50,45,100)
(15,45,200),(50,45,100)
(15,47,200),(50,45,100)
(15,50,200),(30,0,50),(50,45,100)
(15,50,200),(30,1,50),(50,45,100)
(15,50,200),(30,5,50),(50,45,100)
(15,50,200),(30,40,50),(50,45,100)
(15,50,200),(30,45,50),(40,45,60),(50,45,100)
(15,50,200),(30,50,50),(40,45,60),(50,45,100)
(30,190)
(15,50,200),(40,50,60),(50,45,100)
(15,50,200),(40,50,60),(50,50,100)
(15,55,200),(40,54,60),(50,50,100)
(15,55,200),(40,55,60),(50,51,100)
(40,225)
(50,310)
(15,410)
```